

A new kind of parallelism and its programming in the Explicitly Many-Processor Approach

János Végh

Abstract—The processor accelerators are effective because they are working not (completely) on principles of stored program computers. They use some kind of parallelism, and it is rather hard to program them effectively: a parallel architecture by means of (and thinking in) sequential programming. The recently introduced EMPA architecture uses a new kind of parallelism, which offers the potential of reaching higher degree of parallelism, and also provides extra possibilities and challenges. It not only provides synchronization and inherent parallelization, but also takes over some duties typically offered by the OS, and even opens the till now closed machine instructions for the end-user. A toolchain for EMPA architecture with Y86 cores has been prepared, including an assembler and a cycle-accurate simulator. The assembler is equipped with some meta-instructions, which allow to use all advanced possibilities of the EMPA architecture, and at the same time provide a (nearly) conventional-style programming. The cycle accurate simulator is able to execute the EMPA-aware object code, and is a good tool for developing algorithms for EMPA.

Index Terms—computer architecture, processor accelerator, manycore processor, many-processor approach

1 INTRODUCTION

PROGRAMMING hardware accelerators is a real challenge. The accelerator is always outside the processor, and it is efficient because it does not work as the conventional, programmable processors do. Several problems must be solved in order to connect two different worlds: the stored program (von Neumann) processors with rest of the world. It is a challenge for the hardware (HW) designers: the principle of operation of the processing units based on the von Neumann principles has inherent inefficiencies [1], and also using those external accelerators from software running on a conventional architecture is only possible in rather time-consuming ways [2]. It is a challenge also for the software (SW) designers: the program languages are structured according to the von Neumann principles [3], and the programs need to handle facilities, different from the ones, they were designed for. Even the linking method is hard to select. For universal utilization (and also because of the compactness of the Central Processing Unit (CPU)s), the accelerators are usually implemented as input/output (I/O) devices. However, since the OSs must provide protection for the I/O operations, which is a time-consuming procedure [2], only longer code fragments can be delegated to the accelerators.

The modern many-core systems could serve as good starting point to develop general purpose accelerators, using new forms of parallelization, but mainly their programming provokes technical [4], efficiency [5] and performance [6] questions; so that trend seems to be broken [7]. The new kind of parallelism, introduced below, allows to approach the theoretically possible maximal parallelism and at the same time to simplify the HW construction, but of course requires non-conventional processor architecture.

The EMPA architecture [1] seems to be especially hard to program: the processor architecture can be configured by the end-user, and so the architecture may change continuously during the operation; the cores communicate with each other; synchronized internal data transfer between cores takes place; different program parts run in parallel, which must handle data and control dependencies; there are many program counters, belonging to independently running cores; even the conventionally closed unit “machine instruction” can be opened for other processing units, and so on. For the first look, it does not seem possible to program it using facilities mostly similar to the conventional programming. The paper introduces a new kind of parallelism, as well as a programming language and methodology, which allows to utilize the enhanced performance of the EMPA processors, while the programming methods remain as close as possible to the conventional ones.

2 THE DYNAMIC PARALLELISM

The parallelism assumes the presence of several processing units, and the reachable speedup of course strongly depends on the availability of the corresponding HW units. Let us suppose we want to calculate expressions (see [8])

$$A = (C * D) + (E * F)$$

$$B = (C * D) - (E * F)$$

where we have altogether 4 load operations, 2 multiplications, and 2 additions.

2.1 Theoretical (software) parallelism

The theoretical (or SW) parallelism only considers the different kinds of dependences between the values and operations (control and data dependences), and assumes the presence of the needed number of HW units; i.e. it provides a kind of upper bound for the reachable parallelism. For calculating the theoretical parallelism, one can assume that

- J. Végh is with Faculty of Mechanical Engineering and Informatics, University of Miskolc, Hungary
E-mail: J.Vegh@uni-miskolc.hu

Manuscript received July 10, 2016; revised August 26, 2016.

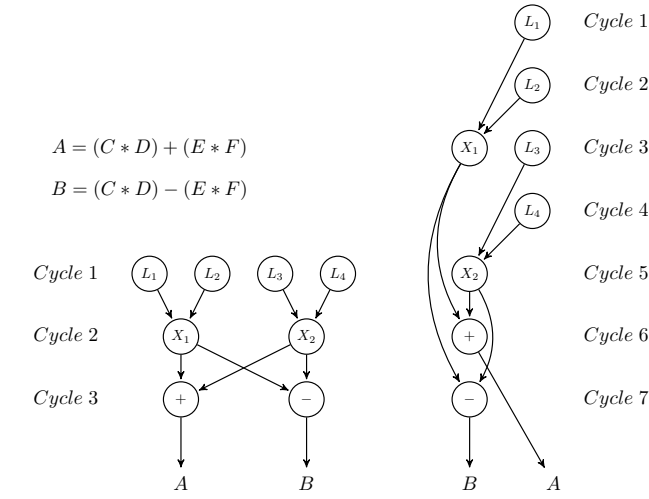


Fig. 1. The theoretical parallelism (left) vs parallelism implemented on a two-issue fixed architecture processor (right)

we have a processor, which has (at least) 4 memory access units, 2 multipliers and 2 adders (or equally: an at least 4-issue universal processor). With such a processor (see Fig. 1) we can load all the four operands in the first machine cycle, to do the multiplications in the second cycle, and to make the addition/subtraction in the last cycle.

2.2 Multiple-issue processor parallelism

The real processors, however, are not built with arbitrarily large number of processing units. In practice, the processors may be built with having so called multiple-issue, single pipeline architecture, i.e. in the same cycle can execute more than one operations, if there are available processing units which are able to perform the requested operation. A so called two-issue processor can make (for example) an arithmetic and a memory access operation at once.

Before making the first multiplication (see Fig 1, right side), in the first two cycles the processor can load the two operands, and in the third cycle, it can make the first multiplication. During the multiplication, the memory access unit is free, so it can load simultaneously the third operand. In the fourth cycle, the fourth operand is loaded, and so finally the second operand for the second multiplication is provided (the first operand is waiting since the third cycle). In the fifth cycle the second multiplication can be performed, and so for the sixth cycle result A is provided, and similarly for the seventh cycle result B is also available. Notice that both the memory access and the arithmetic units are only utilized in 4 cycles (out of the 7), in 3 machine cycles they are unused. Only cycle 3 is when both units are in use.

2.3 Dual core parallelism

One might think that using two independent, single issue processors communicating through shared memories can be equally good for solving the sample task. Initially, both processors can load their arguments (see Fig. 2) and make their multiplication. However, after those operations processors must share their result with their party, i.e. they store their result in the shared memory, and load the result stored by

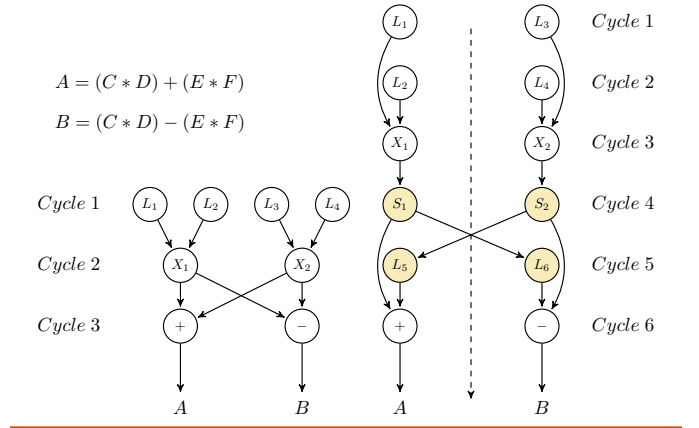


Fig. 2. The theoretical parallelism (left) vs parallelism implemented on a single-issue dual processor system (right).

their party from the shared memory. For doing so, store (S_i) and load (L_i) operations must be inserted in the chain of operations of both processors. These (essentially obsolete, but needed for the communication) operations increase the number of operations to 12, and so both processors must execute 6 cycles. Compare it to the 7 cycles of a 2-issue single-processor system above. Obviously, investing into the second processor and shared memory HW, does not result in the expected increase of performance (in addition, the memory access operations are very expensive in terms of execution time; and also we can only hope that the operand the processor reads was already written by the other party).

2.4 Dynamic parallelism

Both utilizing a limited number of special multiple processing units, and communicating through shared memory degrades the parallelism with respect to the theoretically reachable one. Increasing the number of specialized processing units is possible, but (as Fig. 1 shows) in most of the general purpose cases, those units cannot be fully utilized. Communicating through a shared memory inserts new (obsolete) machine cycles with memory access, and so (as Fig. 2 shows) the number of the cycles needed for executing the task reduces disproportionally. Both solutions strongly limit the available speedup, strongly increase the needed resources and the dissipated power. As an extreme case: the General-Purpose Graphics Processing Unit (GPGPU)s outperform multiple-issue single processor only 2.5 times, although about 100 times better performability is expected [9].

In the first case the cause of the inefficiency is the inflexible architecture, in the second case the lack of any facility of intercore communication. Let us suppose we have a kind of "on demand" type, flexible architecture, i.e. the processor can provide the needed number of processing units for the operation, at the expense of using some time for "renting" the needed unit(s). The rented units are single-issue processors, but they are able to do both memory access and arithmetic operations. In Fig 3 right side, it is assumed that the "cost of renting" is one fifth of a machine cycle. At the very beginning the originating processor (in state O_1) notices that two multiplications shall be performed, so it rents Processing Unit (PU)s H_1 and H_2 , one by one (each

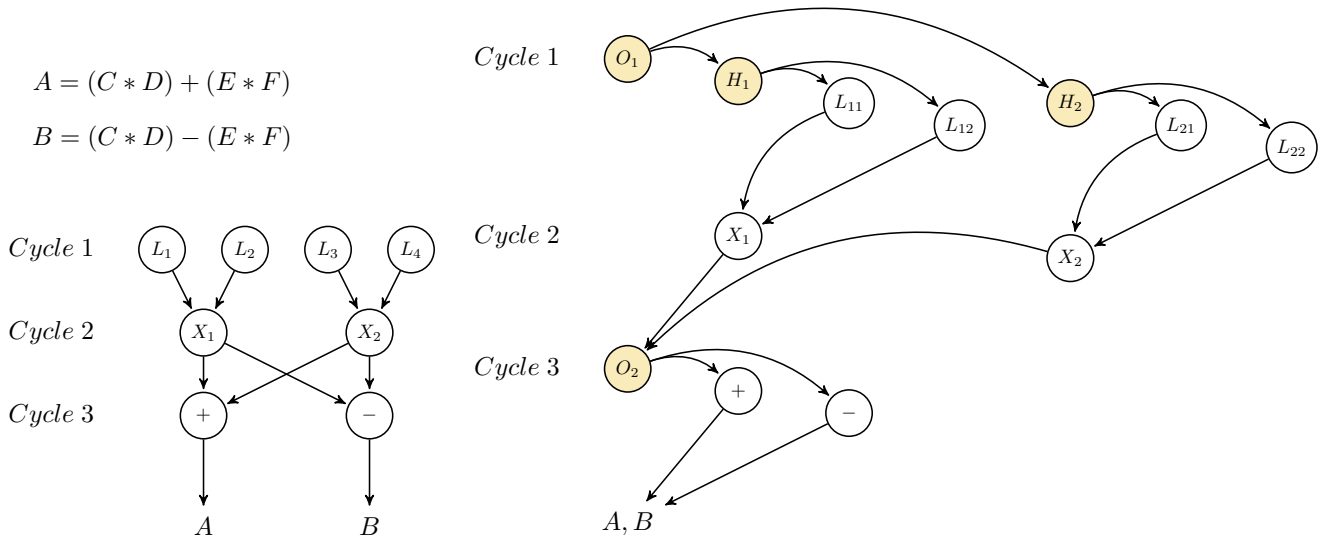


Fig. 3. The theoretical parallelism (left) vs dynamic parallelism implemented on a processor system with runtime configurable architecture (right).

in 0.2 machine cycle) for this goal. Those helper units notice they need two operands to load, so similarly they rent two more processing units only to do the loading.

After loading, the helper processors receive their operands, so they can make the multiplication in their second machine cycle and then deliver the result back to the originating processor, which rents again two more processors for the addition and subtraction, (in the third machine cycle of the originating processor) for the last two operations. After the 3rd machine cycle, both result operands are delivered back to the originating processing unit. The execution time is longer than the theoretical 3 machine cycles. In the figure the total execution time is 3.8 machine cycles, and in the peak period, 6 simple-functionality single-issue processors are used.

Some quantitative parameters of the mentioned parallelization models in the case of calculating our sample expressions are listed in Table 1. Since we have 8 operations, the single-thread execution time is 8. The average degree of parallelization is calculated as the ratio of the number of operations and number of cycles, and the efficiency is given as the ratio of speedup and number of PUs. As it can be expected, this dynamic parallelization model works in a way very similar to the theoretical one, and its degree of parallelization approaches the theoretical one (see Table 1).

TABLE 1
Parameters describing different parallelization models

Parallelism model	Parallelization	N	Speedup	Efficiency
Theoretical	$8/3 = 2.67$	4	$8/3 = 2.67$	$2.67/4 = 0.67$
Two-issue processor	$8/7 = 1.14$	2	$8/7 = 1.14$	$1.14/2 = 0.57$
Dual core	$8/6 = 1.33$	2	$8/6 = 1.33$	$1.33/2 = 0.67$
Dynamic	$8/3.8 = 2.11$	≈ 4.1	$8/3.8 = 2.11$	$2.11/4.1 = 0.51$

2.5 Requirements and consequences of dynamic parallelism

The graph in Fig. 3 right side is essentially the extension of the graph on the left side. The PU is "rented" from some

resource pool and is returned after the operation finished, so in the next machine cycle it can be rented for a different goal. It introduces some (trivial) dependence: before making an operation, a processing unit must be rented; and after the operation finished, it must be released, before starting the next cycle. The renting process is transparent for the originating processing unit, so the original dependence is preserved. The states are in parent-child relationship: a parent can create any number of children, but a child can have only one parent. The parent remains responsible for performing the task it received, but it can delegate part of the task to its children. If the parent has delegated part of its job to the children, it must wait until they terminate.

Since the operation is performed on a different PU rather than the original one, the complete state of the originating internals must be cloned into the created child unit and after finishing the operation, part of the state (the result) must be returned to the parent, in a synchronized way.

In the conventional architectures the machine cycles are uniform. In the dynamic parallelization model the "all children ready" signal triggers the next cycle, which can be somewhat longer, but can also be shorter, if in the child the last internal instruction stage is not utilized. With an effective pre-allocation mechanism, the time needed to allocate a helper core, can approach zero.

2.6 Mapping the operations to processing units

The dynamic parallelism remains "theoretical" in the sense that nothing limits the number of the needed processing units, while in a physical system the number of PUs is limited. The processing graph in Fig. 5 exactly corresponds to the theoretical graph of dynamic parallelism in Fig. 3, the 8th core cannot be utilized by the example code. On a processor having finite number of PUs the processing graph can be compressed horizontally, at the price of increasing the number of the cycles, see Fig. 6. When one keeps the dependence, some operations will simply be postponed for

Listing 1. The dynamic parallelism implemented in assembly language for EMPA/Y86

```

1 | # This is implementing dynamic parallelism, EMPA way
2 | .pos 0 # Program starts at address 0000
3 | 0x000: f5ff72000000 | QCreate QTfinal, %eno # The frame QT
4 | 0x006: f5f62d000000 | QT1multC:QCreate QT1multT, %esi # Return result in %esi
5 | 0x00c: f5f118000000 | QT1loadC:QCreate QT1loadT, %ecx # Load operand 1
6 | 0x012: 501f74000000 | mrmovl C,%ecx
7 | 0x018: f0 | QT1loadT:QTerm # Return operand1 in %ecx
8 | 0x019: f5f625000000 | QT2loadC:QCreate QT2loadT, %esi # Load operand 2
9 | 0x01f: 506f78000000 | mrmovl D,%esi
10 | 0x025: f0 | QT2loadT:QTerm # Return operand1 in %esi
11 | 0x026: flffffff | QWait -1 # Wait until loading finishes
12 | 0x02b: 6316 | xorl %ecx,%esi
13 | 0x02d: f0 | QT1multT:QTerm # Return result in %esi
14 |
15 | 0x02e: f5f755000000 | QT2multC:QCreate QT2multT, %edi # Return result in %edi
16 | 0x034: f5f140000000 | QT3loadC:QCreate QT3loadT, %ecx # Load operand 3
17 | 0x03a: 501f7c000000 | mrmovl E,%ecx
18 | 0x040: f0 | QT3loadT:QTerm # Return operand1 in %ecx
19 | 0x041: f5f74d000000 | QT4loadC:QCreate QT4loadT, %edi # Load operand 4
20 | 0x047: 507f80000000 | mrmovl F,%edi
21 | 0x04d: f0 | QT4loadT:QTerm # Return operand1 in %edi
22 | 0x04e: flffffff | QWait -1 # Wait until loading finishes
23 | 0x053: 6317 | xorl %ecx,%edi
24 | 0x055: f0 | QT2multT:QTerm # Return result in %edi
25 | 0x056: flffffff | QWait -1 # Wait until second finishes
26 | # Now the operands are in %edi and %esi
27 | 0x05b: f5f663000000 | QT3addC :QCreate QT3addT, %esi
28 | 0x061: 6076 | addl %edi,%esi
29 | 0x063: f0 | QT3addT :QTerm # Return operand1 in %esi
30 | 0x064: f5f76c000000 | QT3subC :QCreate QT3subT, %edi
31 | 0x06a: 6167 | subl %esi,%edi
32 | 0x06c: f0 | QT3subT :QTerm # Return operand1 in %edi
33 | 0x06d: flffffff | QWait -1 # Wait until third cycle finishes
34 | 0x072: f0 | QTfinal: QTerm # All calculations finished
35 | 0x073: 00 | halt
36 |
37 | # Array of 4 elements
38 | 0x074: | .align 4
39 | 0x074: 06000000 | C: .long 0x6
40 | 0x078: 02000000 | D: .long 0x2
41 | 0x07c: 03000000 | E: .long 0x3
42 | 0x080: 01000000 | F: .long 0x1
43 |

```

a later machine cycle, prolonging the processing time and decreasing the reached parallelism. The PUs are “rented” strictly for the time of performing the processing step, so after a while “reprocessed” PUs get available.

Obviously, the traditional fixed architectures are not able to adapt themselves to the task executed, so for that a special architecture [1] must be used, which has some extra signals, storages and functionality, see Fig. 4. Such an architecture can be implemented using methods known in reconfigurable technology, like using block RAMs, configurable wiring between fixed functionality blocks, etc.

Also, to program such a task special programming instructions are needed. The code producing the processing diagrams (see section 6.2) in Figs. 5-6 is shown in Listing 1.

Technically, one ‘higher level’ core is needed, which embeds the code calculating the expressions in the sample. The individual machine instructions are put in Quasi-Thread (QT) frames [1] only to provide complete visual analogy with Fig. 3. The same core could start reserving a helper core to load one operand; while waiting, the core itself could load the another operand, and make the operation itself. The used method demonstrates, however, that this kind of parallelism can be extended towards both elementary operations (like individual machine instructions) and complex expression evaluations (like the 8 elementary operations in the example code). The operations in all cases can be independently executed, and when discovering parallelism, no HW limitations shall be considered.

Here an event-controlled, rather than clock-controlled, operation takes place, much similarly to the pipelining and hyperthreading. This operating principle is not foreign from the Neumann paradigms: there a new operation can only start when the old operation frees the PU.

3 THE TOOL CHAIN FOR UTILIZING DYNAMIC PARALLELISM

3.1 The goal of the programming tools

The EMPA architecture not only provides a flexible HW for implementing dynamic parallelism (actually: provides an end-user programmable architecture), but also provides other forms of acceleration, like replacing certain machine instruction sequences with using inter-core operating signals and replacing (apparently non-parallelizable) operations with inter-core cooperation. Those facilities are unusual in the conventional programming, so a special programming approach had to be developed. That approach must use conventional terms and programming interface, to give chance to use higher-level languages for programming the EMPA architecture, and at the same time must provide a way to fully utilize the unconventional features of EMPA.

3.2 The Y86 processor

EMPA actually means some architectural principles, rather than a certain concrete processor or core. The work described here is based on using the Y86 [2] processor as core. It is not a real processor in the sense that it has very few instructions (finally, its purpose is educational). From our particular point of view it has advantages, like

- models a widely used architecture
- it is simplified and without optimization
- it has an open-source toolchain (simulator and assembler)
- its Instruction Set Architecture (ISA) allows to implement additional instructions and registers with easy

As the first steps towards a tool chain, an EMPA-aware ISA simulator and an assembler has been prepared [11]. These simple tools allow to prepare executable programs for the EMPA and characterize the performance features of the architecture [1], as well as to develop and scrutinize further features.

For educational purposes, a simplified Intel X86 processor has been developed and made publicly available [2], including an ISA-level simulator and an assembler. Using a more advanced (non-educational) processor simulator would considerably extend the usability of the simulator. However, those simulators are optimized for an absolutely different single-processor architecture, and also usually do not provide an easy path to adding the needed extensions. It is only fair to compare EMPA to an unoptimized conventional processor. If the comparison is advantageous for EMPA, similar, but different HW accelerators will be developed for this architecture, too, and those optimized architectures can again be fairly compared to the optimized conventional architectures.

3.3 Extensions to the Y86 ISA

The original Y86 ISA [2] utilizes one-byte instruction codes, and in an unused instruction slot the group of meta-instructions utilized to configure the newly introduced supervisor (SV) control layer [1] of the processor has been implemented. Following the Y86 conventions, a member of the EMPA metainstruction group is coded as the group code in the high nibble and the member code in the low nibble. The mnemonic of a metainstruction always starts with a 'Q' (for QT). The metainstructions can have zero, one, or two arguments, and their total length is between one and six bytes.

3.4 The EMPA simulator(s)

The simulator was written having electronic components in mind, i.e. it operates in a cycle-accurate way. The engine uses as core functionality the Y86 ISA simulator, slightly extended. Both a command line based and a Qt5 [12] based graphical interfaces have been added to the simulator. The GUI simulator is equipped with step-wise execution and logging; it produces processing diagrams like the one in Fig. 7, and provides different kinds of statistics, allowing to scrutinize the sophisticated operation of the EMPA/Y86 processor, and to derive operational characteristics [1].

4 ASSEMBLY EXTENSIONS FOR EMPA

The support for the unconventional EMPA features is implemented through a surprisingly low number of new assembly (meta)instructions and other extensions, causing just a little difference relative to the traditional single-processor case.

4.1 Creating and terminating quasi-threads

The EMPA-aware code is organized into special units of QTs [1], of intermediate size and structure, somewhere between the HW unit 'machine instruction' and SW unit 'thread'. Handling QTs is supported by the new assembly instructions **QCreate** and **QTerm**, which must be used in a bracket-like way.

The **QCreate** instruction has two arguments. The first one is the address of the matching **QTerm**. This informs the parent core, where to continue after delegating the code in the QT. The second argument is the *link register* (either a physical one or a pseudo-register, see section 4.5).

The **QTerm** instruction has no arguments, but implies a **QWait -1** and clones back the link register, defined by the matching **QCreate**.

Example (As shown, extensive labeling is utilized for referencing):

```
CLabel: QCreate TLabel, %eax
... executable instructions ...
TLabel: QTerm
```

4.2 Synchronizing QTs

The assembler provides two kinds of instructions to support explicit waiting. Instruction **QWait** only considers its own children, while **QPWait** considers the sisters (the other children of the parent). Upon finding a **QxWait**, the requesting

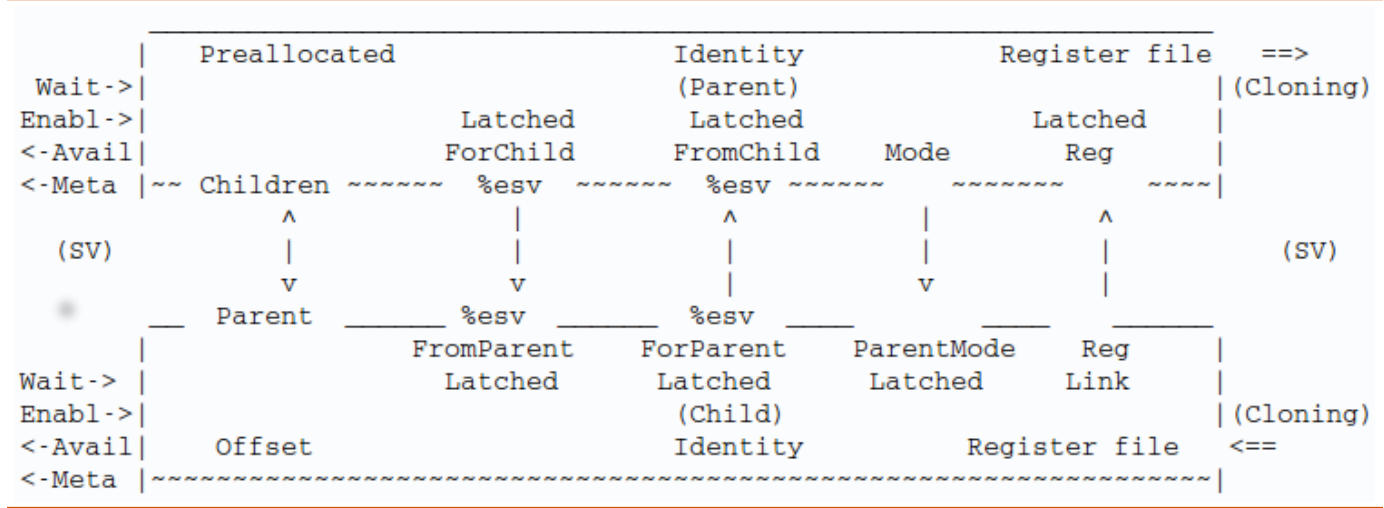


Fig. 4. The communication signals and data between parent and child cores in EMPA

core gets blocked until the referred to QT terminates. The wait instructions have one argument. The argument is either the address of one particular QT or a wildcard (All QTs in the scope).

TABLE 2
The trigger signals and their effect on latched inter-core data communication

Trigger	Transfer in parent	Transfer in child
QxCreate	ForChild→ register file→ Mode→	→FromParent →register file →ParentMode
QxWait (QTerm)	link register←	←link register
QTerm	ForChild→	←FromParent
mass processing (mode dependent)	FromChild→	←ForParent

The **QxWait** metainstructions play an important role also in synchronizing data transfer between parent and child cores. As described in [1], the asynchronous operation of cores needs well-designed transport policy, especially when using the link register. Table 2 shows the triggered data exchange policy between the parent and child cores.

Example :

```
QWait CLabel # Wait specific QT
QWait -1 #Wait all sister QTs
```

4.3 Subroutine call with EMPA

The instruction **QCall** provides a possibility to place the called QT out of the body of the code. The instruction has an address argument, where the called QT begins. The referred to QT commences in the child core, the control in the parent returns immediately to the address next to **QCall**. Since the argument of the metainstruction is the address of a metainstruction **QCreate**, its functionality is automatically implied in functionality of **QCall**. Practically, the difference is that the called QT code is located outside the body of the parent control flow, resulting in a modular, clear program structure.

Notice that the return address, unlike in Single Processor Approach (SPA) case, should *not* be remembered: the called

subroutine runs on a new core and uses its own Program Counter (PC), while the caller continues processing with instruction next to **QCall**. The HW should not save the return address, in this way less memory cycles and machine instructions needed, and also the HW addresses are not interlaced in the SW-handled stack items. This also reduces the need for calling frames and simplifies addressing of automatic or passed variables.

Example:

```
QCall CLabel
```

4.4 Supporting cooperation between cores

Several classes of processor accelerators serve executing masses of instructions in parallel. EMPA provides mass processing modes for this goal. Mass processing functionality is implemented through the metainstructions **QAlloc**, **QCreate** and **QFCreate**.

The two arguments of **QAlloc** are a mode value, and a register, containing the argument for the requested operation. Since **QAlloc** actually is a request to the SV that the requestor core wants to rent additional core(s), the program must prepare for two answers, and handle them in two different branches. The two branches are implemented by metainstructions **QCreate** and **QFCreate**. Exactly one of them will be executed after the last **QAlloc**, the other party will behave as a NOP instruction (i.e. it follows an **if..then..else** logic).

Both cases must be programmed: there is one QT prepared for the case when mass processing in the requested mode is possible, and another one for the case when not. These two metainstructions have the arguments and functionality identical with those of **QCreate**, except that they are only executed if the pre-allocation was successful and was NOT successful, respectively.

Metacommand **QAlloc** must also assure that the needed number of cores will be available at some later time for the parent core. For this goal, the needed cores are preallocated for the core: they will appear for the concurrently working cores as unavailable cores, but **QCreate** can use them to start new QTs.

If the core preallocation was successful (i.e. there are enough cores), the **QTCreat** branch will be executed as many times as needed. This is controlled by the SV, and is carried out in consecutive clock cycles, one by one. The PC of the parent will advance to the address next to the matching **QTerm** only when **QTCreat** was processed the requested number of times, in certain modes each time with a newly allocated child core. In this way the **QTCreat** actually corresponds to as many **QCreate** metainstructions as many repetitions were requested by **QAlloc**. In these actions only the preallocated (rather than unallocated) cores are used.

The functionality of **QTCreat** is not simply making a replica of the parent for the preallocated cores. The pseudo-register **%esv** behaves as configured by the 'mode' argument of **QAlloc**. The **QFCreat** is just a wrapper for the instructions and metainstructions to be executed when there are not enough cores for the given type of mass processing, i.e. the processing must follow some other way. These conditional allocations can be nested. Note that the link register for both branches must be the same.

Example:

```

QAlloc mode, %edx
Q4TC: QTCreat Q4TT, %ecc
    instructions if we have enough cores
Q4TT: QTerm
Q4FC: QFCreat Q4FT, %ecc
    instructions if we do NOT have enough
cores
Q4TT: QTerm

```

4.5 Pseudo-registers

For implementing an effective data transfer between the cooperating cores, some pseudo-registers have been implemented. The pseudo-registers are seen by the ISA as registers, but they represent not a simple storage. Rather, they might behave in an extraordinary way: they can transfer data synchronously between parent and child, in both directions, and they can change the data they provide for their partner between the consecutive accesses.

Register **%eno** is used where the syntax requires the presence of a register argument, but the related activity is not desirable. Register **%ecc** is for returning condition codes only, while **%esv** is used for parent-child related activity. While the first two pseudo-registers can only be used as arguments of **QCreate** (i.e. as link registers), the functionality of **%esv** largely depends on the context it is used in, see Table 3.

TABLE 3
The context dependent mapping of register **%esv** to latched registers

Context	As source	As destination
Cloning (link register)	ForParent	FromChild
Child in mass processing	FromParent	ForParent
Parent in mass PREprocessing	FromParent	ForChild
Parent in mass POSTprocessing	FromChild	ForParent
Other (general) case	FromChild	ForParent

Fig. 4 (repeated from [1] for convenience) shows how the parent and child cores communicate with each other using latch registers. The **%esv** register is mapped in a context-dependent way to the latched registers, see Table 3. As shown in the Table, the mass processing parent role is divided into PRE-processing (between **QAlloc** and **QTCreat**), and POST-processing (between **QTerm** of the child and **QTerm** of the parent) phases. Using **%esv** as link register, the SV reads the content of 'ForParent' in the child and writes it to 'FromChild' in the parent. This means, that if a child core wants to transfer to its parent the data it received from its own child, the child core must use an explicit **rmovl %esv, %esv** instruction. Register **%esv** is designed for helping cooperation, and cannot be used as general purpose register.

5 ALGORITHMIC ASPECTS

It was early recognized [3], that even our programming languages are heavily influenced by the single-processor approach, and so are our algorithms. The disclosed new possibilities in the EMPA architecture also need new thinking in designing the algorithms. The synergy between the possibilities of EMPA and the new EMPA-aware algorithms (i.e. suggesting methods to implement in EMPA which can simplify or boost old or develop more efficient new algorithms) can result in further performance increase of our HW/SW systems. EMPA provides a couple of general frames and methods for using such possibilities, as shown by the examples below, and is ready to implement further such frames. Below, a simple programming example is presented in four different versions, to illustrate how different accelerating principles [1] of EMPA can be used in practice.

5.1 The conventional coding (or NO mode mass processing)

The first mode is the NO mass processing mode. It exactly matches the traditional programming: NO real mass processing takes place, no metainstructions are used and the required loop control functionality is provided through calculations. It requires only the original PU, uses the same instructions and has the same execution time, as the traditional programs.

In this code the operands are loaded immediately before the calculation. The summing is as simple as possible: first the sum is cleared (Listing 2, line 6), and the number of items verified (Listing 2, line 7). These are one-time actions, not parallelized.

From beginning with "Loop", the usual activity takes place: in addition to the payload operation (Listing 2, line 10) **addl %esi, %eax**, using non-payload operations the item is addressed, fetched (Listing 2, line 9), the address advanced to the next item (Listing 2, line 12), the loop counter advanced and verified (Listing 2, line 14), and a conditional jump instruction (Listing 2, line 15) closes the loop. Upon exiting the loop, register **%eax** contains the sum (Listing 2, line 16).

Notice that register **%eax** contains the partial sum during the calculation. This is the main source of inefficiency: in the payload operation **addl %esi, %eax** the previous

Listina 2. The sum-up routine using NO EMPA facilities

```

1      | # This is summing up elements of vector
2      | .pos 0 # Program starts at address 0000
3      | 0x000: 30f206000000 | irmovl $4,%edx # No of items to sum
4      | 0x006: 30f134000000 | irmovl array,%ecx # Array address
5      |
6      | 0x00c: 6300 | xorl %eax,%eax # sum = 0
7      | 0x00e: 6222 | andl %edx,%edx # Set condition codes
8      | 0x010: 7332000000 | je End
9      | 0x015: 506100000000 | Loop: mrmovl (%ecx),%esi # get *Start
10     | 0x01b: 6060 | addl %esi,%eax # add to sum
11     | 0x01d: 30f304000000 | irmovl $4,%ebx #
12     | 0x023: 6031 | addl %ebx,%ecx # Start++
13     | 0x025: 30f3ffffff | irmovl $-1,%ebx #
14     | 0x02b: 6032 | addl %ebx,%edx # Count--
15     | 0x02d: 7415000000 | jne Loop # Stop when 0
16     | 0x032: 00 | End: halt
17     |
18     | # Array of 4 elements
19     | 0x034: | .align 4
20     | 0x034: 0d000000 | array: .long 0xd
21     | 0x038: c0000000 | .long 0xc0
22     | 0x03c: 000b0000 | .long 0xb00
23     | 0x040: 00a00000 | .long 0xa000

```

Listina 3. The sum-up routine using EMPA looping FOR facilities

```

1      | # This is summing up elements of vector
2      | # Uses loop organization facilities of EMPA
3      | 0x000: | .pos 0 # Program starts at address 0000
4      | 0x000: 30f206000000 | irmovl $4, %edx # No of items to sum
5      | 0x006: 30f124000000 | irmovl array, %ecx # Array address
6      |
7      | 0x00c: 6300 | xorl %eax, %eax # sum = 0
8      | 0x00e: f4f201 | QAlloc 1, %edx # allocate %edx times a helper core
9      | 0x011: 201d | rrmovl %ecx, %esv # Overwrite with array address
10     | 0x013: f6f021000000 | QT1LoopC:QTCreate QT1LoopT, %eax
11     | 0x019: 501d00000000 | mrmovl (%esv), %ecx # This is the value
12     | 0x01f: 6010 | addl %ecx, %eax # Add it to the collected sum
13     | 0x021: f0 | QT1LoopT:QTerm
14     | 0x022: 00 | halt
15     |
16     | # Array of 4 elements
17     | 0x024: | .align 4
18     | 0x024: 0d000000 | array: .long 0xd
19     | 0x028: c0000000 | .long 0xc0
20     | 0x02c: 000b0000 | .long 0xb00
21     | 0x030: 00a00000 | .long 0xa000

```

content of the destination register is read, then the operation performed and the result is written back as new content to the destination register. Notice that the non-payload instructions have no role at all for the calculation, furthermore that we are interested in the final result only, and not at all in the partial results.

5.2 The basic loop: FOR mode mass processing

As seen above, in such a simple loop the non-payload activities require much more executable instructions, than the payload activity; and so: they take most of the processing

time. The overall performance can be enhanced through omitting those service instructions as machine instructions, and use HW-implemented facilities instead. EMPA provides simple loop organization facility, which helps to eliminate those non-payload instructions.

The first three machine instructions (Listing 3, lines 4-7) are identical with those shown in listing 2. The metainstruction `QAlloc 1, %edx` (Listing 3, line 8) sets operating mode 1 (this is FOR), preallocates one core and tells SV it wants to use the pre-allocated core `%edx` (actually: 4) times for looping. This core will be available for the requesting

core only, and only until looping is over.

The SV clears in the parent the base index in 'ForChild' to be transferred to the rented child. This value will be incremented by 4 between the iterations, so the actual child can always reach the actual offset value. Since `%ecx` contains the base address of the vector, and the child inherits the register file of the parent, the child could calculate the actual address from the base and the offset. However, it takes time, so the pseudo-register `%esv` provides a useful facility to shorten the code.

In the pre-processing phase of the loop (between `QAlloc` and `QTCreat`, see Table 3) writing `%esv` (Listing 3, line 9) means writing into 'ForChild'. So, the instruction `rrmovl %ecx, %esv` writes the base address into 'ForChild' in the parent. Since the contents of 'ForChild' is increased by 4 between iterations, the child receives a ready-made address, there is no need to make address calculation.

The metainstruction `QTCreat QTLoopT, %eax` (Listing 3, line 10) will create child QT. The SV keeps the core pre-allocated until loop terminates. In this mode PC of the parent core remains pointing to `QTCreat` (Listing 3, line 10) while the loop is running. In the following clock periods, the parent must wait, since the QT running on the child core is not yet terminated. When the child QT finally terminates and the SV notifies the parent, it immediately executes the next iteration, until the iteration count reached.

The payload activity is done by the child core, i.e. by instructions between `QTCreat` (Listing 3, line 10) and (Listing 3, line 13). Here the core fetches the actual argument (Listing 3 7) from the address given by contents of its own pseudo-register `%esv`. In this mode reading `%esv` corresponds to reading 'FromParent' (see Table 3), which receives its contents from 'ForChild' in the parent when `QTCreat` is executed.

The child core inherits the internals (including contents of register file) of its parent when the QT is created, and returns the content of its link register to the corresponding register in the parent when `QTerm` is executed, see Table 2. I.e. on entry (Listing 3 8) `%eax` contains the previous partial sum, on exit `%eax` contains the new partial sum, which will be cloned back to the parent, and serves as the old partial sum in the next iteration.

Although not used in the present example, to provide a possibility to `break` out of the loop, the child can write its own pseudo-register `%esv`, which means writing into 'ForParent'. The child can write 0 into that register. Upon executing `QTerm`, the contents of that register will be written in register 'FromChild' in the parent. Before executing `QTCreat`, the SV checks 'FromChild' in the parent, and terminates loop if it is cleared, otherwise executes `QTCreat` again: increases the address in 'ForChild' and decreases count in 'FromChild'. Of course, at the beginning `QAlloc` writes the requested number of repetitions into 'FromChild'.

Notice that the complete loop organization is accomplished by the SV, on behalf of the parent core. The child's kernel can do any, much more complex activity. The only limiting factor is that only the content of the link register is back cloned to the parent. Also notice that here actually no parallelization occurs. The parent is waiting when the child is processing, and always only one child is used. Another variant for FOR functionality is to reserve a core for the

individual vector elements. The child cores, as they would be created in adjacent cycles in that mode, would receive the correct address from the parent. However, after termination they would overwrite `%eax`, or would have to wait the termination of the previous QT without performance gain. Because of this, that mode cannot be used for summing up elements of a vector. However, EMPA has a more elegant and useful method for solving that problem.

5.3 The specialized loop: SUMUP mode mass processing

In summing up elements of a vector, the partial sum must be written back into a register in a machine instruction, and read out the same again in the next cycle. *It is because the atomic unit in SPA is the machine instruction.* Since for the time of looping a persistent connection can exist between the parent and its children, EMPA can provide a way to eliminate this weakness, using its SUMUP mode.

The first three executed instructions are the same as in case of Listing 2. The metainstruction `QAlloc` (Listing 4, line 8) now sets mode 5, and `%edx` now contains the number of requested helper cores (i.e. this time we want to use several cores in parallel, rather than one core several times). To spare time, the next instruction (Listing 4, line 9) overwrites the offset address passed to the child with the base address of the array, exactly, as in the case of FOR mode. Also the same, that PC in the parent will stay pointing to `QTCreat` (Listing 2, line 10) and creating children, one after the other. It is, however, different, that several cores are preallocated at the beginning, so the parent shall not wait: in the consecutive cycles will create children, every time in a new core, which child core will work in parallel with each other child cores and the parent core.

The payload instructions (i.e. instructions between `QTCreat` (Listing 4, line 10) and `QTerm` (Listing 4, line 13) are very similar to the case of FOR mode. The important difference is that now the partial sum is 'stored' in register `%esv`. In this mass processing mode writing `%esv` means *sending the data to a prepared adder in the parent* [1], where the addition is immediately executed, rather than reading the previous partial sum from a temporary storage and writing it back. (Technically, it is written in 'ForParent' in the child, but the instruction triggers copying the summand to 'FromChild' in the parent, which is connected to one of the inputs of the adder, while the other input is connected to the output of the adder.) Both the old and new partial sums are only stored in the adder circuit, rather than in some special register. The child QTs are created with one clock cycle delay relative to each other, so they will send their fetched data also with the same delay for the parent, so the adder can receive the data and execute the addition without waiting or queuing.

The parent and its children will run in parallel, and after starting the last child, the parent will continue with the instruction at address next to `QTerm` (Listing 4, line 14). At that time some of the children might still work, so here a `QWait` metainstruction must be used, otherwise the adder might contain not the final sum. When all children

Listina 4. The sum-up routine using EMPA looping SUMUP facilities

```

1      | # This is summing up elements of vector, EMPA way
2      | # Will sum elements secretly, using mass processing
3      0x000:      | .pos 0 # Program starts at address 0000
4      0x000: 30f206000000 | irmovl $4,%edx # No of items to sum
5      0x006: 30f12c000000 | irmovl array,%ecx # Array address
6      |
7      0x00c: 6300 | xorl %eax,%eax # sum = 0
8      0x00e: f4f205 | QAlloc 5, %edx # Preallocate %edx cores
9      0x011: 201d | rrmovl %ecx, %esv # Overwrite with array address
10     0x013: f6ff21000000 | QTLLoopC:QTCreat QTLLoopT, %eno # %esv sums up values
11     0x019: 501d00000000 | mrmovl (%esv),%ecx # get *Start + Index
12     0x01f: 601d | addl %ecx, %esv # Sum up in parent's %esv
13     0x021: f0 | QTLLoopT:QTerm
14     0x022: f113000000 | QWait QTLLoopC # Wait until child ready
15     0x027: 20d0 | rrmovl %esv, %eax # Make the result visible
16     0x029: 00 | halt
17     |
18     | # Array of 4 elements
19     0x02c:      | .align 4
20     0x02c: 0d000000 | array: .long 0xd
21     0x030: c0000000 | .long 0xc0
22     0x034: 000b0000 | .long 0xb00
23     0x038: 00a00000 | .long 0xa000

```

terminated, the parent will be in post-processing phase, so reading `%esv` results in reading 'FromChild' which latches the output of the adder. The instruction `rrmovl %esv, %eax` (Listing 4, line 15) will bring to light the till invisible sum. Note, that in this mode the link register has no role, so `%eno` is used in QT creation (Listing 4, line 10).

5.4 The adaptive processing

When using `QAlloc` (Listing 5, line 8), the successful execution is not granted at all. The compiler cannot know in advance, how many cores will be available when the metainstruction will be executed, from having as many cores as vector elements, to having one core only, so it must prepare for all possible cases. The metainstructions `QTCreat` and `QFCreat` provide an `if...then...else` construction to solve this problem. It means that the compiler must generate code for all those cases, and the SV chooses one according to the actual core availability. As it will be shown in the example, these constructions can be nested.

This adaptive program is displayed in Listing 5. Actually, the kernels of the three previous programs are put together into a special structure. The most performable operating mode for summing up elements of a vector is SUMUP mode, provided that there are enough cores available at the moment when the summation must be executed. If the first `QTCreat` (Listing 5, line 10) is not successful, then a `QFCreat` (Listing 5, line 16) is executed. Within this latter block another `QTCreat` (Listing 5, line 21-25) QT pair is located. If less than 4 cores are available, then the program attempts to allocate at least one more extra core (i.e. attempts to use the next, less performable, but also less resource-hungry operating mode). If this also fails, then continues with the conventional processing.

As shown, the compiler accounts for all possibilities, and the SV chooses the right code fragment. Anyhow: the program will run to completion, but its execution time will depend on the actual availability of cores. This availability is not sensible for the external observer, only the different execution times (see Table 4) will be noticed. I.e. the multicore EMPA processor might appear as a super-power single-core processor for the user.

TABLE 4
Speedup and effective parallelization for adaptive SUMUP (see Listing 5), in function of the number of available cores

Time (clocks)	No of cores (k)	Speedup (s)	α_{eff}
142	1	1	1
156	2	0.91	-0.20
156	3	0.91	0.65
80	4	1.58	0.58
38	5+	3.74	0.92

In Table 4, the execution time of program in Listing 2 serves as the base of comparison. The rest of the lines in the table show cases when during executing the program given in Listing 5 the processor has different number of available cores. The slight increase relative to row 1 in execution time is due to executing the metainstructions: this is the price one has to pay for running a program, designed for many-core systems, on a single-core system. As shown in Table 4, in a system having 5 cores, this summing is nearly 4 times quicker (as shown in Fig 7, only a small fragment of the code is parallelized). The column α_{eff} [13] also shows, that EMPA is designed for many-cores: the utilization efficiency increases with the increasing number of cores used. A more detailed analysis of performance of EMPA is given in [1].

Listina 5. The sum-up routine using adaptive EMPA looping facilities

```

1      | # This is summing up elements of vector, combined EMPA way
2      | # Will sum elements as actual core allocation allows
3      0x000:      | .pos 0 # Program starts at address 0000
4      0x000: 30f204000000 | irmovl $4,%edx # No of items to sum
5      0x006: 30f170000000 | irmovl array,%ecx # Array address
6
7      0x00c: 6300 | xorl %eax,%eax # sum = 0
8      0x00e: f4f205 | QAlloc 5, %edx # Preallocate %edx cores
9      0x011: 201d | rrmovl %ecx, %esv # Overwrite with array address
10     0x013: f6ff21000000 | QTLLoopC:QTCreat QTLLoopT, %eno # %esv sums up values
11     0x019: 501d00000000 | mrmovl (%esv),%ecx # get *Start + Index
12     0x01f: 601d | addl %ecx, %esv # Sum up in parent's %esv
13     0x021: f0 | QTLLoopT:QTerm
14     0x022: f113000000 | QWait QTLLoopC # Wait until children ready
15     0x027: 20d0 | rrmovl %esv, %eax # Make the result visible
16     0x029: f7f069000000 | QFLoopC:QFCreate QFLoopT, %eax # %esv sums up values
17     | # No 4 cores, check if we have at least 1 more
18     0x02f: f4f201 | QAlloc 1, %edx # Preallocate 1 core %edx times
19     0x032: 201d | rrmovl %ecx, %esv # Overwrite with array address
20     0x034: 6300 | xorl %eax,%eax # sum = 0 ; array address is still OK
21     0x036: f6ff44000000 | QT1LoopC:QTCreat QT1LoopT, %eno
22     0x03c: 501d00000000 | mrmovl (%esv), %ecx # This is the value
23     0x042: 6010 | addl %ecx, %eax # Add it to the collected sum
24     0x044: f0 | QT1LoopT:QTerm
25     0x045: f7f068000000 | QF1LoopC:QFCreate QF1LoopT, %eax
26     | # We are alone, make normal processing
27     0x04b: 506100000000 | Loop: mrmovl (%ecx),%esi # get *Start
28     0x051: 6060 | addl %esi,%eax # add to sum
29     0x053: 30f304000000 | irmovl $4,%ebx #
30     0x059: 6031 | addl %ebx,%ecx # Start++
31     0x05b: 30f3ffffff | irmovl $-1,%ebx #
32     0x061: 6032 | addl %ebx,%edx # Count--
33     0x063: 744b000000 | jne Loop # Stop when 0
34     0x068: f0 | QF1LoopT:QTerm
35
36     0x069: f0 | QFLoopT:QTerm
37     0x06a: f1ffffff | QWait -1 # Wait until everything finished
38     0x06f: 00 | halt
39
40     | # Array of 4 elements
41     0x070: | .align 4
42     0x070: 0d000000 | array: .long 0xd
43     0x074: c0000000 | .long 0xc0
44     0x078: 000b0000 | .long 0xb00
45     0x07c: 00a00000 | .long 0xa000

```

6 THE SIMULATOR

The toolset has been published [11] with online documentation. Because of the permanent development, it is still in alpha quality, but it is usable. The unconventional features need careful utilization.

6.1 The command line simulator

The command line version of the simulator runs to completion, and makes extensive logging in a file showing all the details of the operation. Although it is very useful during debugging, and is inevitable when a power user attempts

to “fine tune” his program, for educational and demonstrational purposes a Qt5 [12] based graphical simulator has also been prepared. On the screen the complete internal life of the EMPA architecture is displayed, as the cores are rented, the intercommunication of the cores, etc. The execution (in step-wise or run modes) can be followed.

6.2 The processing diagram

The *processing diagram* is a by-product of the simulators and it attempts to visualize the rather sophisticated internal operation of the EMPA processor. The diagram should show, at which time, by which core, which instruction was executed;

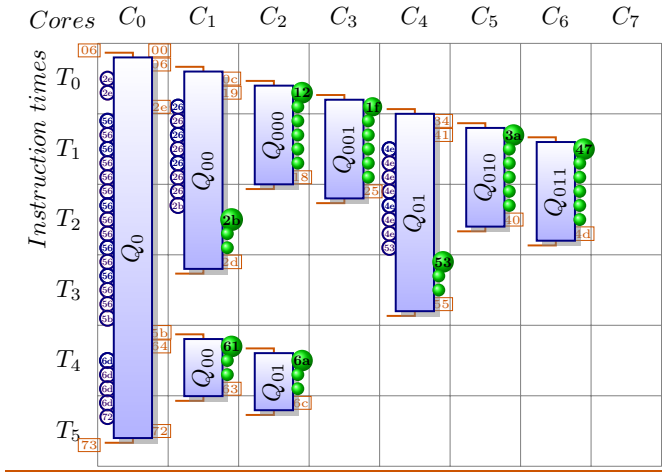


Fig. 5. The processing diagram (compare to Fig. 3) of the sample program, running on an 8-core EMPA processor

and how the cores interacted with each other; the cores can execute conventional executable or metainstructions. So, a lot of information must be crowded into the figure.

The diagram shows the cores on the horizontal axis and the time on the vertical one. For better orientation, grid lines are put to every 5th clock cycle. The clock cycle here is the length of an SV operation, the instruction execution is supposed to be of variable length. Arbitrary, but reasonable instruction lengths are assumed.

The rectangular blocks represent the QTs, with hooks at the top and bottom, for their creation and termination. In the columns C_x the vertical rectangles represent the "lifetime" of a QT. At the times outside the QT rectangles, the core is in power economy mode, not running a QT.

The parent-child relationship is illustrated with the labels of the QTs: the first few chars are identical with those of the parent, and the last char denotes the sequence number of the child. On the figure (as well as in the simulator log files), for the human reader, core sequence numbers and textual QT ID strings are shown rather than the "one-hot" bitmasks used by the simulator.

The memory address of a metainstruction is shown on the right side of the QT in a rectangle, the address of an executable command is shown on the top of a bigger ball, and some smaller balls represent the duration of the instruction. While a core is waiting, at the corresponding time a circle with the respective memory address is displayed at the left side of the QT blocks. From the memory address the source code can be found using the listings. Accessing pseudo register `%esv` of parent by children is marked by an angular bracket, also showing the direction of the transferred data. The places where summands are sent for their parent for summation, are marked by an extra plus character.

6.2.1 Dynamic parallelism

The processing graph (see Fig. 5) is derived from the theoretical dependencies, so the memory accesses within the cycles have no dependency on each other; i.e. the memory can be accessed without limitation, no need for assuring coherence, i.e. no need to use slow, power hungry and

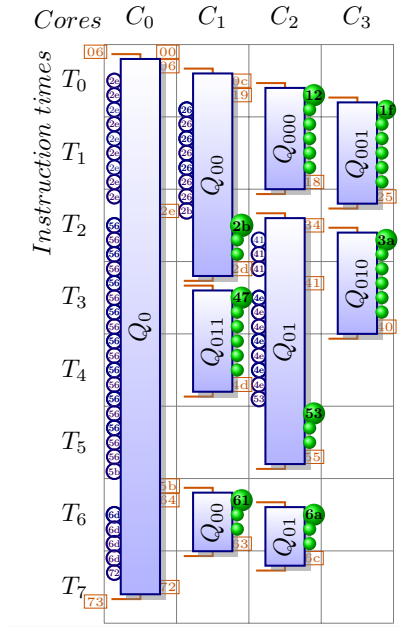


Fig. 6. The processing diagram (compare to Figs. 3 and 5)) of the sample program, running on an 4-core EMPA processor

expensive shared memory. A memory of type like [10] with several independent ports can solve the task.

The dynamic parallelism remains "theoretical" in the sense that nothing limits the number of the needed processing units, while in a physical system the number of PUs is limited. The processing graph in Fig. 5 exactly corresponds to the theoretical graph of dynamic parallelism in Fig. 3, the 8th core cannot be utilized by the example code.

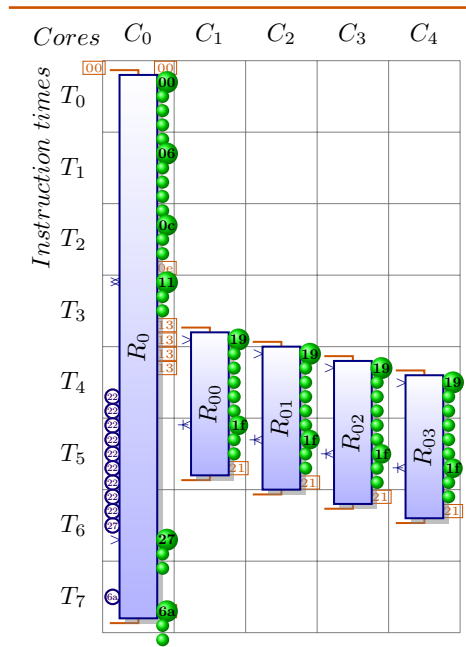


Fig. 7. Processing diagram of execution program in Listing 5 running on 5 cores

On a processor having finite number of PUs the processing graph can be compressed horizontally, at the price of

increasing the number of the cycles, see Fig. 6. That diagram shows how the same code will be executed in a system with 4 PUs only. The first load operation can be started immediately, the second loading only when some PU is released from the first operation. The SV in EMPA notices that core C_2 finished the processing and is free again, so the state H_2 is mapped to C_2 rather than to C_4 as in the case of 8 cores. Anyhow, the operation takes place when both operands are available.

Some operations will simply be postponed for a later machine cycle, prolonging the processing time and decreasing the reached parallelism. The programmer can give the theoretical dependency independently of the HW, and the processing graph will adjust itself to the HW at runtime.

6.2.2 Vector sumup

Fig. 7 visualizes the operation of the EMPA processor, when running program shown in Listing 4 on a 5-core system. Notice how the inter-core operations (receiving the address of the operand and sending the summand to the parent) are shifted in time, and the final sum remains latent until an explicit parent instruction takes it out into a “visible” register.

7 CONCLUSIONS

The presented programming methodology demonstrates, how the dynamic parallelism and other unusual features of the EMPA architecture can be supported in a programming style, which is powerful enough to use the performance increasing facilities of the architecture and at the same time remains close enough to the conventional programming style. This promises good hopes that the EMPA architecture could be effectively supported from high-level languages.

In the EMPA implementation, a special purpose “ad-hoc” computing unit is assembled on the fly, just for the time of the (maybe complex) operation. This computing unit works with the maximum reachable parallelism, using the needed minimum of computing resources, with maximum efficiency. The operation may be simple like loading an operand or computing the two expressions used in discussing types of parallelisms, or complex like summing up elements of a vector. The introduced extra signals and local storages allow to omit some instructions used traditionally to organize a loop, and replace them with using internal signals. The close vicinity of the PUs (like in the case of modern many-core processors) allows using cooperative regime for making calculations, and allows to parallelize even the classic non-parallelizable sumup operation. This latter mode enables to gain an order of magnitude in speedup. The programming facilities allow the programmers (person or compiler) to use the unusual facilities offered by EMPA, using traditional terms and tools. Although the dependencies shall be correctly considered, the hardware conditions should not be known at the time of coding: the architecture follows the prescribed logic of programming, but adapts its resource need to the momentary HW availability.

The real-time characteristics of processors also benefit from EMPA. To service an interrupt, no state saving and restoring is needed, saving memory cycles and code. The program execution will be predictable: the processor need

not be stolen from the running main process. The atomic nature of executing QTs will prevent issues like priority inversion, eliminating the need for special protection protocols.

From the point of view of accelerators, an EMPA processor provides a natural interface for linking special accelerators to the processor. Any circuit, being able to handle data and signals shown in Fig. 4 can be linked to an EMPA processor with easy.

REFERENCES

- [1] J. Végh, “A configurable accelerator for manycores: the Explicitly Many-Processor Approach,” *ArXiv e-prints*, Jul. 2016.
- [2] R. E. Bryant and D. R. O'Hallaron, *Computer Systems: A Programmer's Perspective*. Pearson, 2014.
- [3] J. Backus, “Can Programming Languages Be liberated from the von Neumann Style? A Functional Style and its Algebra of Programs,” *Communications of the ACM*, vol. 21, pp. 613–641, 1978.
- [4] J. Larus, “Programming Multicore Computers,” *Communications of the ACM*, vol. 58, no. 5, p. 76, May 2015.
- [5] A. Mahesri, N. J. Wang, and S. J. Patel, “Hardware Support for Software Controlled Multithreading,” *SIGARCH Comput. Archit. News*, vol. 35, no. 1, pp. 3–12, Mar. 2007.
- [6] N. Satish, C. Kim, J. Chhugani, H. Saito, R. Krishnaiyer, M. Smelyanskiy, M. Girkar, and P. Dubey, “Can Traditional Programming Bridge the Ninja Performance Gap for Parallel Computing Applications?” *Commun. ACM*, vol. 58, no. 5, pp. 77–86, Apr. 2015. [Online]. Available: <http://doi.acm.org/10.1145/2742910>
- [7] U. Vishkin, “Is Multicore Hardware for General-Purpose Parallel Processing Broken?” *Communications of the ACM*, vol. 57, no. 4, p. 35, May 2014.
- [8] K. Hwang and N. Jotwani, *Advanced Computer Architecture: Parallelism, Scalability, Programmability*, 3rd ed. Mc Graw Hill, 2016.
- [9] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, and P. Dubey, “Debunking the 100x gpu vs. cpu myth: An evaluation of throughput computing on cpu and gpu,” in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ser. ISCA '10. New York, NY, USA: ACM, 2010, pp. 451–460. [Online]. Available: <http://doi.acm.org/10.1145/1815961.1816021>
- [10] Cypress, “CY7C026A: 16K x 16 Dual-Port Static RAM,” <http://www.cypress.com/documentation/datasheets/cy7c026a-16k-x-16-dual-port-static-ram>, 2015.
- [11] J. Végh, “EMPAThY86: A cycle accurate simulator for Explicitly Many-Processor Approach (EMPA) computer,” jul 2016. [Online]. Available: <https://github.com/jvegh/EMPAThY86>
- [12] J. Blanchette and M. Summerfield, *A Brief History of Qt. C++ GUI Programming with Qt 4*. Prentice-Hall, 2006.
- [13] J. Végh, P. Molnár, and J. Vászárhelyi, “A figure of merit for describing the performance of scaling of parallelization,” *CoRR*, vol. abs/1606.02686, 2016. [Online]. Available: <http://arxiv.org/abs/1606.02686>



VÉGH, János received his PhD in Physics in 1991, ScD in 2006. He is working in computing since the early 80's, and since 2008 is a full professor in Informatics, currently with University of Miskolc, Hungary. In research, he is looking for ways to prepare more performant and more predictable computing, especially using mainly multi-core processors. He is also dealing with soft processors and hardware-assisted operating systems, reconfigurable and many-processor computing.